Fall, 1997 Name: _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 2.3, 3, 4.5–6

This test has 7 questions and pages numbered 1 through 6.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 8pt font) of notes. No photo-reduction is permitted. These notes are to be handed in at the end of the test. Use of other notes or failure to follow these instructions will be considered cheating.

During the test, if you need more space for an answer, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For programs, indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. We will take off a small amount if you do not give `TYPE` comments for recursive helping procedures. However, you do not have to write such comments for procedures for which the type is stated in the problem. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Subset of Scheme You May Use

Unless otherwise stated in a problem, when solving problems you may only use standard features of the language that we discussed in class, `define-record`, `variant-case`, and helping functions that you define yourself. The standard is defined by the *Revised*^{4} *Report on the Algorithmic Language Scheme*.

Parts of Scheme You May *Not* Use

Unless otherwise stated in a problem, you are prohibited from defining your own macros, and from using internal defines, all the input and output facilities, and the following keywords and procedures. (Don't worry if you don't know what these are.)

```
call-with-current-continuation  do
```

1. (10 points) In each of the spaces provided ("_____") below, write, in set brackets, the entire set of the bound variables in the corresponding Scheme expression. For example, write $\{v, w\}$, if the bound variables are v and w. If there are no bound variables, write $\{\}$. (You're supposed to know what a "bound variable" is.)

(a)
```
(let ((xp1 (add1 x))
      (y (plus xp1 1)))
  (grow (plus xp1 y) (plus y 10)))
```
_____

(b)
```
(letrec ((o (lambda (n) (let ((b (e n))) b)))
         (e (lambda (x) (if (z? x) 1 (o (minus x one))))))
  (o (e 3)))
```
_____

2. (5 points) In the space provided ("_____") below, write, in set brackets, the entire set of the free variables in the corresponding Scheme expression. For example, write $\{v, w\}$, if the free variables are v and w. If there are no free variables, write $\{\}$. (You're supposed to know what a "free variable" is.)

```
(letrec ((o (lambda (n) (let ((b (e n))) b)))
         (e (lambda (x) (if (z? x) 1 (o (minus x one))))))
  (o (e 3)))
```

_____

3. (5 points) In the following expression, draw an arrow from each bound ⟨varref⟩ to its declaration.

```
(lambda (x)

  (lambda (f y g)

    (lambda (z g)

      (x (f z) (g y)))))
```

4. (10 points) Consider the expression in the previous problem again. Give the lexical address form of that expression, by filling in the blanks below, replacing all the ⟨varref⟩s in the above expression by their lexical addresses. (You're supposed to know what a "lexical address" is.)

```
(lambda (x)
  (lambda (f y g)
    (lambda (z g)
      _____________________________________________________ )))
```

5. (30 points) Write a procedure, `syntax-expand` of the following type

```
(-> (parsed-exp) parsed-exp)
```

that desugars short-circuit `and` expressions into `if` expressions.

In this problem, the variant record type `parsed-exp`, is defined by the union of the following record types, which forms an abstract syntax. As we have been doing, we use the numeric literal 0 for false and 1 for true.

```
(define-record varref (var))
(define-record lit (datum))
(define-record app (rator rands))
(define-record if-exp (test-exp then-exp else-exp))
(define-record and-exp (left right))
```

Your procedure is to return a `parsed-exp` with the same behavior as its argument, but which does not contain any `and-exp` records. We expect you to understand this desugaring, but there are a few examples on the next page.

The following are examples for the problem on the previous page.

```
(syntax-expand (make-and-exp (make-varref 'x) (make-varref 'y)))
 ==> #(if-exp #(varref x) #(varref y) #(lit 0))
(syntax-expand
 (make-and-exp
  (make-app (make-varref 'equal) (list (make-varref 'x) (make-lit 7)))
  (make-app (make-varref 'less) (list (make-varref 'y) (make-varref 'z)))))
 ==> #(if-exp #(app #(varref equal) (#(varref x) #(lit 7)))
              #(app #(varref less) (#(varref y) #(varref z)))
              #(lit 0))
(syntax-expand
 (make-app
  (make-varref 'f)
  (list (make-and-exp
         (make-app (make-varref 'equal) (list (make-varref 'x) (make-lit 7)))
         (make-and-exp (make-varref 'y) (make-varref 'z))))))
 ==> #(app #(varref f)
           (#(if-exp #(app #(varref equal) (#(varref x) #(lit 7)))
                     #(if-exp #(varref y) #(varref z) #(lit 0))
                     #(lit 0))))
```

6. (5 points) Fill in the blank lines in the following transcript.

```
> (define states (cons 'iowa (cons 'nebraska '())))
> (define more (cons 'colorado states))
> (set! states (cons 'utah states))
> more

> states

> (set-car! (cdr states) 'kansas)
> states

> (cadr more)
```

7. (25 points) This is a problem about transforming a procedural representation of geometric regions to a record representation.

   As helpers, we will use a type `point` and a procedure `distance` that gives the distance between points.

```
(define-record point (x y))
(define distance ; TYPE: (-> (point point) number)
  (lambda (p1 p2)
    (let ((x-diff (- (point->x p1) (point->x p2)))
          (y-diff (- (point->y p1) (point->y p2))))
      (sqrt (+ (* x-diff x-diff)
               (* y-diff y-diff))))))
```

   The following is the procedural representation of geometric regions.

```
(define circle   ; TYPE: (-> (point number) region)
  (lambda (center radius)
    (lambda (p)
      (<= (distance center p) radius))))
(define outside  ; TYPE: (-> (region) region)
  (lambda (r)
    (lambda (p)
      (not (is-in? r p)))))
(define intersect ; TYPE: (-> (region region) region)
  (lambda (r1 r2)
    (lambda (p)
      (and (is-in? r1 p) (is-in? r2 p)))))
(define is-in?     ; TYPE: (-> (region point) boolean)
  (lambda (r p)
    (r p)))
```

   For example, if we define

```
(define origin (make-point 0 0))
(define circle2 (circle origin 2))
```

   then the following are examples of how the above code works:

```
(is-in? circle2 origin) ==> #t
(is-in? circle2 (make-point 4 -20)) ==> #f
(is-in? (intersect circle2 (outside circle2)) origin) ==> #f
(is-in? (intersect circle2 (circle origin 1)) (make-point 0.1 0.5)) ==> #t
```

   Your task is to transform the above representation of regions into one that uses records. Do this by giving the `define-record` declarations needed, and filling in the bodies of the procedures. There are places for both of these on the next page. You must use `variant-case` in your solution.

```
;;; write the define-record declarations below



(define circle    ; TYPE: (-> (point number) region)
  (lambda (center radius)


(define outside   ; TYPE: (-> (region) region)
  (lambda (r)


(define intersect  ; TYPE: (-> (region region) region)
  (lambda (r1 r2)


(define is-in?      ; TYPE: (-> (region point) boolean)
  (lambda (r p)
```